

```
using System;
using System.Threading;
using Microsoft.DirectX.DirectSound;
using Microsoft.Win32.SafeHandles;
using System.Windows.Forms;
using System.Diagnostics;

namespace SoundCapture
{
    /// <summary>
    /// Base class to capture audio samples.
    /// </summary>
    public abstract class SoundCaptureBase : IDisposable
    {
        const int BufferSeconds = 3;

        // NotifyPoints determines how many samples are passed into the processData method
        // 8 times a second for gameplay
        // 2 times a second for tuning screen
        // This is because the more samples the fft is given, the more accurate it is
        int NotifyPointsInSecond;

        // change in next two will require also code change
        const int BitsPerSample = 16;
        const int ChannelCount = 1;

        int sampleRate = 44100;

        bool isCapturing = false;
        bool disposed = false;

        public bool IsCapturing
        {
            get { return isCapturing; }
        }

        public int SampleRate
        {
            get { return sampleRate; }
            set
            {
                if (sampleRate <= 0) throw new ArgumentOutOfRangeException();

                EnsureIdle();

                sampleRate = value;
            }
        }

        Capture capture;
        CaptureBuffer buffer;
        Notify notify;
        int bufferLength;
        AutoResetEvent positionEvent;
        SafeWaitHandle positionEventHandle;
        ManualResetEvent terminated;
        Thread thread;
        Device device;

        public SoundCaptureBase(Device device, int NotifyPointsInSecond)
        {
            this.device = device;
            this.NotifyPointsInSecond = NotifyPointsInSecond;

            positionEvent = new AutoResetEvent(false);
            positionEventHandle = positionEvent.SafeWaitHandle;
            terminated = new ManualResetEvent(true);
        }

        private void EnsureIdle()
        {
            if (IsCapturing)
```

```

        throw new SoundCaptureException("Capture is in process");
    }

    /// <summary>
    /// Starts capture process.
    /// This function sets up the capture buffer
    /// Computes the points in the capture buffer to trigger the process data method
    /// Creates a sound capture thread that calls runs the threadLoop function
    /// Starts the capture thread
    /// </summary>
    public void Start()
    {
        EnsureIdle();

        isCapturing = true;

        // Buffer format setup
        WaveFormat format = new WaveFormat();
        format.Channels = ChannelCount;
        format.BitsPerSample = BitsPerSample;
        format.SamplesPerSecond = SampleRate;
        format.FormatTag = WaveFormatTag.Pcm;
        format.BlockAlign = (short)((format.Channels * format.BitsPerSample + 7) / 8);
        format.AverageBytesPerSecond = format.BlockAlign * format.SamplesPerSecond;

        bufferLength = format.AverageBytesPerSecond * BufferSeconds;

        CaptureBufferDescription description = new CaptureBufferDescription();
        description.Format = format;
        description.BufferBytes = bufferLength;

        // sets the specified user device
        capture = new Capture(device.GetID());

        // creates the capture buffer
        buffer = new CaptureBuffer(description, capture);

        int waitHandleCount = BufferSeconds * NotifyPointsInSecond; // total positions in the buffer to be computed
        BufferPositionNotify[] positions = new BufferPositionNotify[waitHandleCount]; // array of buffer notification position
        for (int i = 0; i < waitHandleCount; i++)
        {
            BufferPositionNotify position = new BufferPositionNotify();
            position.Offset = (i + 1) * bufferLength / positions.Length - 1; // working out next position offset

            // assigning the event to be triggered when the position in the buffer is reached
            position.EventNotifyHandle = positionEventHandle.DangerousGetHandle();

            positions[i] = position;
        }

        notify = new Notify(buffer);
        // Summary:
        //     Sets the notification positions for triggering events during capture or playback.
        // Parameters:
        //     notify:
        //     An array of Microsoft.DirectX.DirectSound.BufferPositionNotify structures
        //     that describe the notification positions.
        notify.SetNotificationPositions(positions); // passed in the computed positions

        terminated.Reset();

        // create the sound capture thread
        thread = new ThreadStart(ThreadLoop);
        thread.Name = "Sound capture";
        thread.Start();
    }

    private void ThreadLoop()
    {
        // Start capturing data into the buffer.
        buffer.Start(true);
    }

```

```

try
{
    int nextCapturePosition = 0;
    WaitHandle[] handles = new WaitHandle[] { terminated, positionEvent };

    //MessageBox.Show("Thread Started");

    //MessageBox.Show(Convert.ToString(handles.Length));

    //Waits for any of the elements in the specified array to receive a signal.
    //Returns a Int32 set to the index of the element in waitHandles that received a signal.
    while (WaitHandle.WaitAny(handles) > 0)
    {
        // The waitAny method waits for terminated and positionEvent to be called
        // ie the first buffer notify position has been reached

        int capturePosition, readPosition;
        buffer.GetCurrentPosition(out capturePosition, out readPosition);

        //MessageBox.Show("Current CapturePosition: " + capturePosition);
        //MessageBox.Show("Current readPosition: " + readPosition);

        int lockSize = readPosition - nextCapturePosition;
        if (lockSize < 0) lockSize += bufferLength;
        if ((lockSize & 1) != 0) lockSize--;

        int itemCount = lockSize >> 1;

        //MessageBox.Show("Current nextCapturePosition: " + nextCapturePosition);

        short[] data = (short[])buffer.Read(nextCapturePosition, typeof(short), LockFlag.None, itemCount);

        //MessageBox.Show("Buffer Length: " + data.Length);

        // Pass the capture data into the Process Data method
        // The process data method applys the FFT algorithm
        // and then finds the fundamental frequency in the frequency spectrum
        ProcessData(data);

        nextCapturePosition = (nextCapturePosition + lockSize) % bufferLength;
    }
}
finally
{
    buffer.Stop();
}

}

/// <summary>
/// Processes the captured data.
/// </summary>
/// <param name="data">Captured data</param>
protected abstract void ProcessData(short[] data);

/// <summary>
/// Stops capture process.
/// </summary>
public void Stop()
{
    if (isCapturing)
    {
        isCapturing = false;

        terminated.Set();
        thread.Join();

        notify.Dispose();
        buffer.Dispose();
        capture.Dispose();
    }
}

void IDisposable.Dispose()

```

```
{
    Dispose(true);
}

~SoundCaptureBase()
{
    Dispose(false);
}

private void Dispose(bool disposing)
{
    if (disposed) return;

    disposed = true;
    GC.SuppressFinalize(this);
    if (IsCapturing) Stop();
    positionEventHandle.Dispose();
    positionEvent.Close();
    terminated.Close();
}
}
```